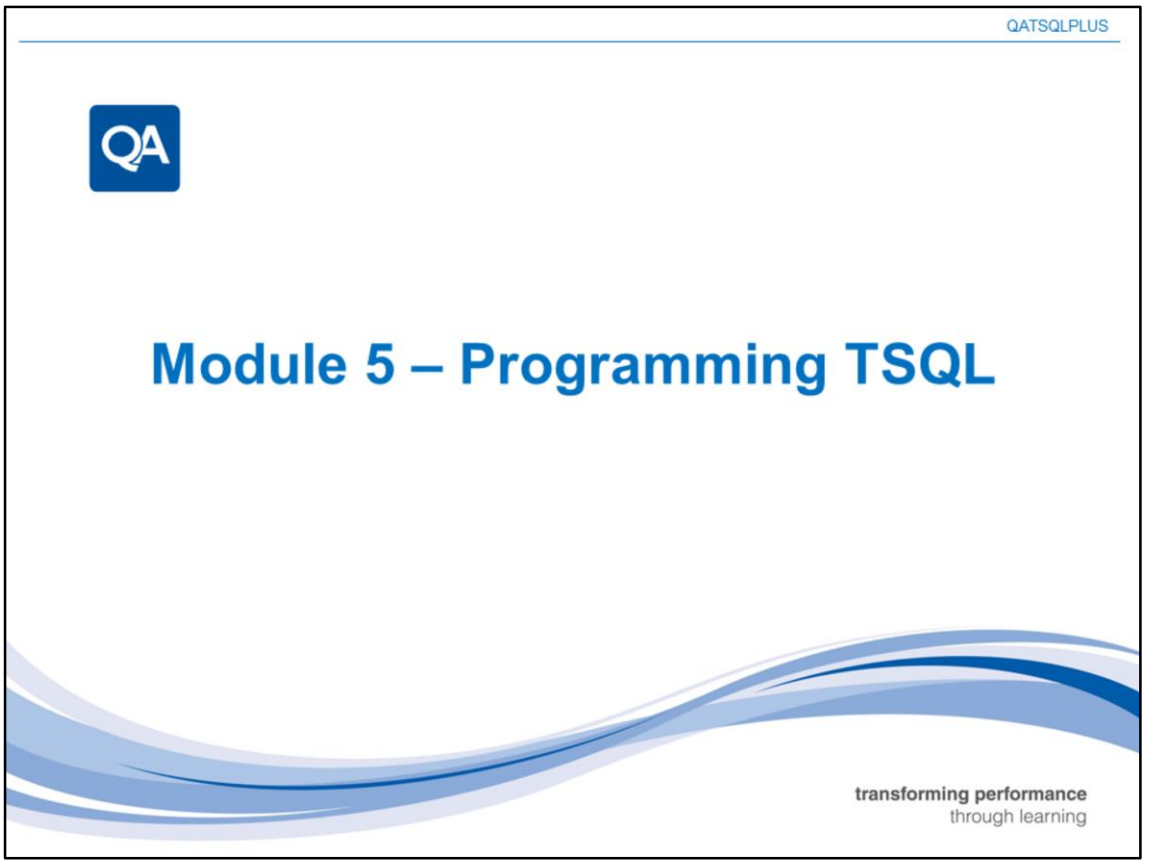
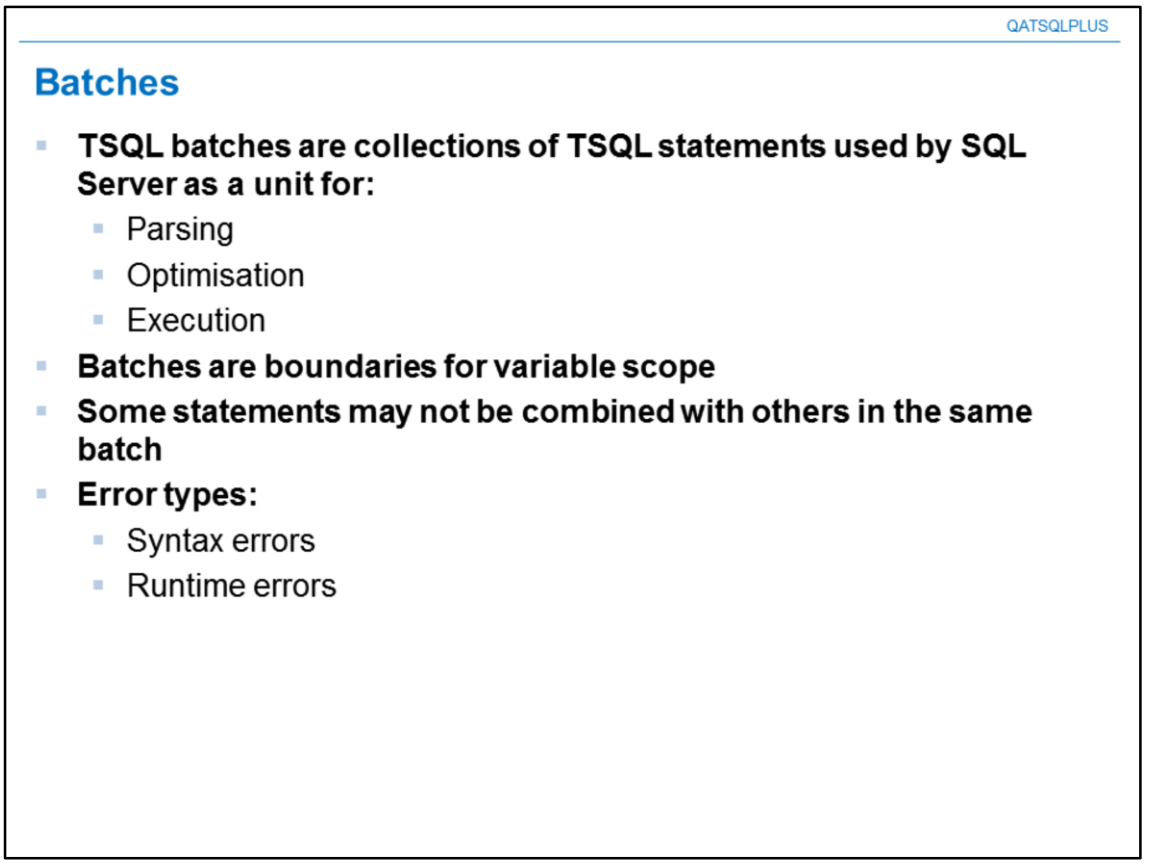

QATSQLPLUS
Module 5 – Programming TSQL
transforming performance
through learning

QATSQLPLUS

Programming TSQL

- Batches
- Variables
- If / Else
- Begin / End
- While loops
- Dynamic SQL
- Cursors
- Return



Batches

- TSQL batches are collections of TSQL statements used by SQL Server as a unit for:
    - Parsing
    - Optimisation
    - Execution
- Batches are terminated with GO by default
- Batches are boundaries for variable scope, so any declared variable will be removed at the end of a batch
- Error types:
    - **Syntax errors** – the code is rejected as the batch is assessed together, such as variable not being in scope
        - Subsequent batches will execute
    - **Runtime errors** – by default the rest of the code in a batch will attempt to run, such as divide by zero errors
        - Subsequent batches will execute

Example

The code below will output 1 and 4, with a syntax error for the undeclared variable @X. As each batch is assessed and executed, batch 1 executes, batch 2 does not execute at all due to a syntax error but execution continues with batch 3.

Batch 1

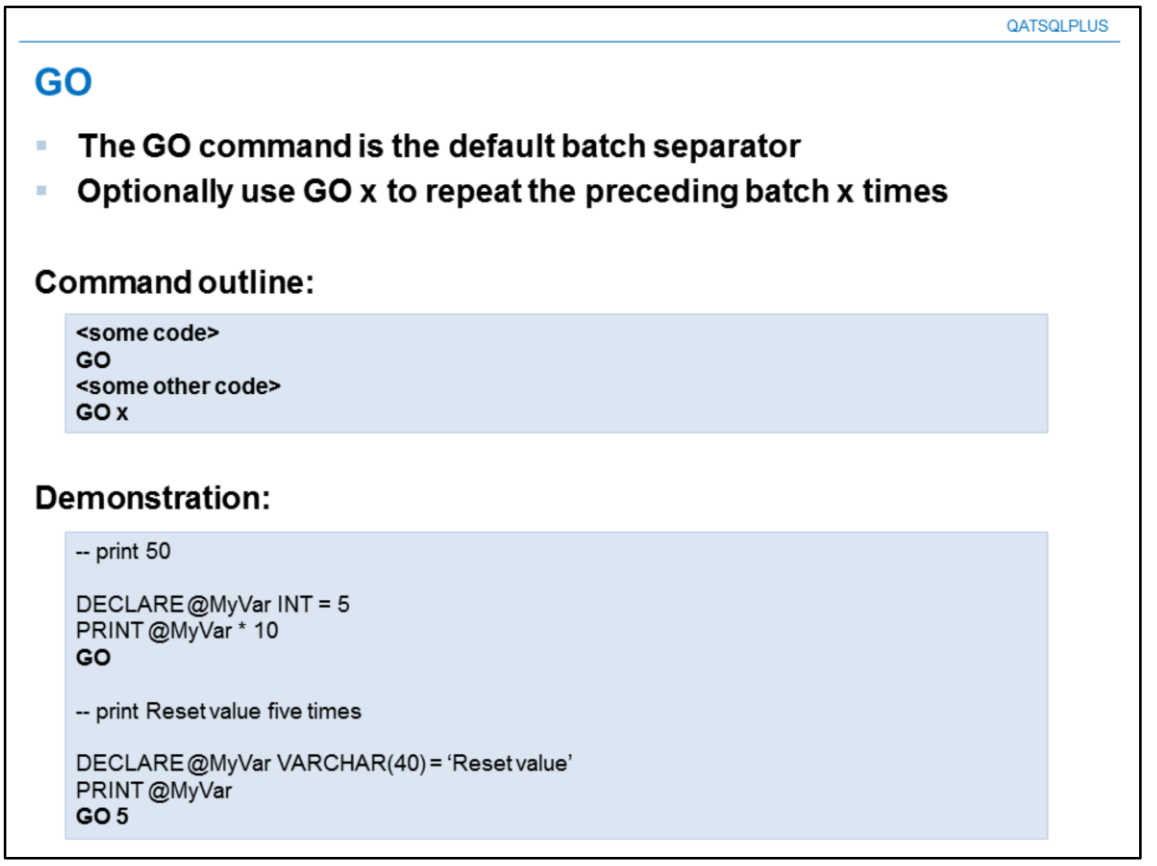
```
PRINT '1'
GO
```

Batch 2

```
PRINT ‘2'
PRINT @X
PRINT ‘3'
GO
```

Batch 3

```
PRINT ‘4'
GO
```



GO

- GO is not a TSQL statement, it is a command recognised by SQL utilities (SQLCMD, OSQL, ISQL and SSMS)
- GO is a signal to the SQL utility to send the current section of code as a batch
- The batch separator command can be changed to another word using the SSMS properties
- Starting at SQL Server 2008, the GO statement can have a number, to execute the batch a set number of times
- Certain statements in TSQL require to be in a batch without any other code, such as CREATE VIEW and CREATE PROC
    - So a GO could be placed before and after the statement to place in its own batch, within a script holding multiple queries

Batches

Some statements need to be separated, use the batch separator statement, GO by default

```
CREATE VIEW dbo.SalesCategorySummary AS
    SELECT Category, Quarter, SUM(Sales) AS TotalSales
        FROM SalesSource
        GROUP BY Category, Quarter
            WITH CUBE
GO
CREATE VIEW dbo.PivotQuarters AS
    SELECT *
        FROM (SELECT Category, Quarter, Sales FROM SalesSource) AS A
        PIVOT
        (SUM(Sales) FOR Quarter IN ([Q1],[Q2],[Q3])) AS B
GO
```

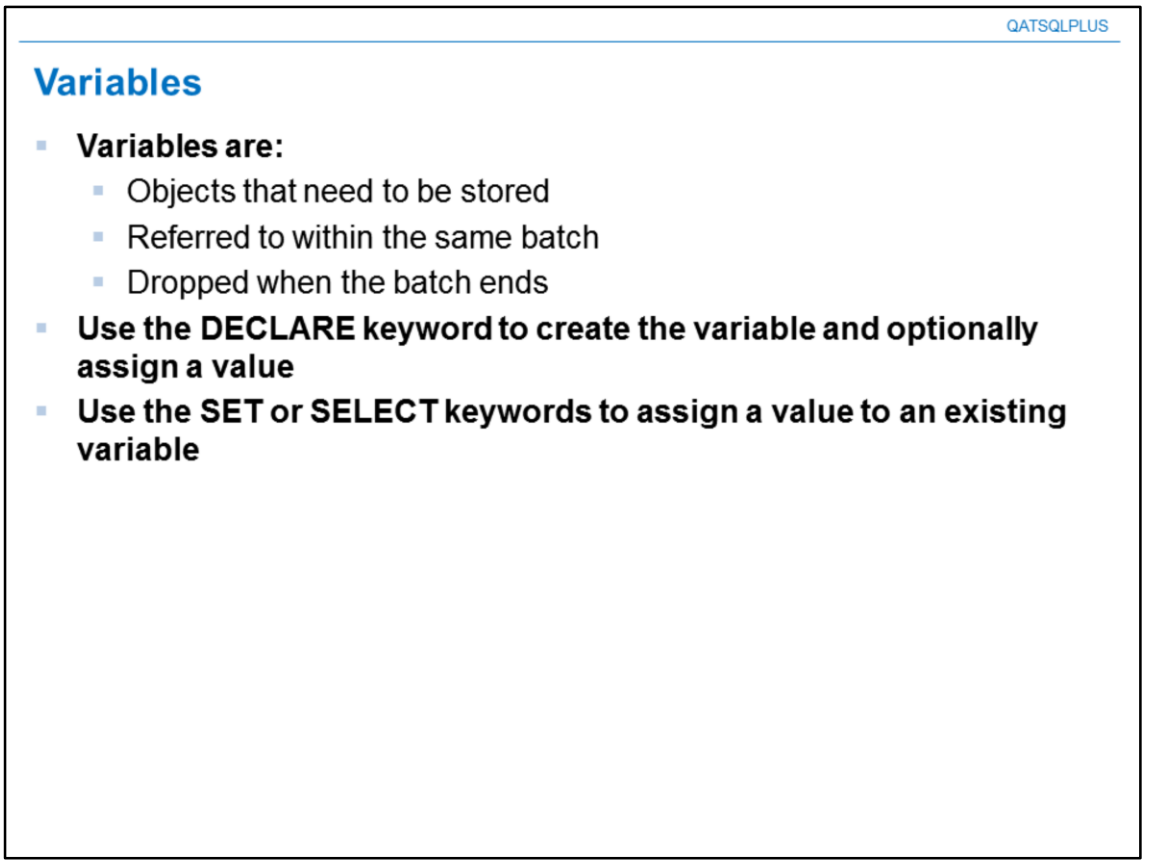
Batches (cont.)

Invalid batch does not run due to syntax errors

```
-- DestinationTable has two columns (ID INT, TextString VARCHAR(40))
INSERT INTO DestinationTable VALUES (1,'One')
INSERT INTO DestinationTable VALUES (2,'Two','Another Value') -- batch rejected due to 3 columns being presented
SELECT * FROM DestinationTable
```

Runtime error

```
-- DestinationTable has two columns (ID INT, TextString VARCHAR(40))
INSERT INTO DestinationTable VALUES (1,'One')
INSERT INTO DestinationTable VALUES ('Two','Minutes') -- batch fails at this step at runtime but steps before ran
SELECT * FROM DestinationTable
```



Variables

- Variables are objects that need to be stored and referred to within the same batch, and are dropped when the batch ends
- Use the DECLARE keyword to create the variable and optionally assign a value
    - DECLARE @A INT
    - DECLARE @B INT = 25

- Use the SET or SELECT keywords to assign a value to an existing variable, usually
    - Use SET when setting a not based on a dataset
    - Use SELECT when setting a value from a dataset

Examples

```
DECLARE @NumberOfVendors INT
SELECT @NumberOfVendors = COUNT(*)
   FROM dbo.Vendors
PRINT @NumberOfVendors
```

Declaring Variables

- **Variables are declared starting with an @ sign**
- **There are many data types that can be used:**

Integers	Decimals	Date/Time	Character	Others
Bit TinyInt SmallInt Int BigInt	Decimal Numeric Money SmallMoney Float Real	DateTime DateTime2 DateTimeOffset SmallDateTime Date Time	Char Varchar	XML Text Binary Varbinary Image UniqueIdentifier Geography Geometry Table Cursor

- **Some developers declare Alias data types within their databases which as based on existing data types**

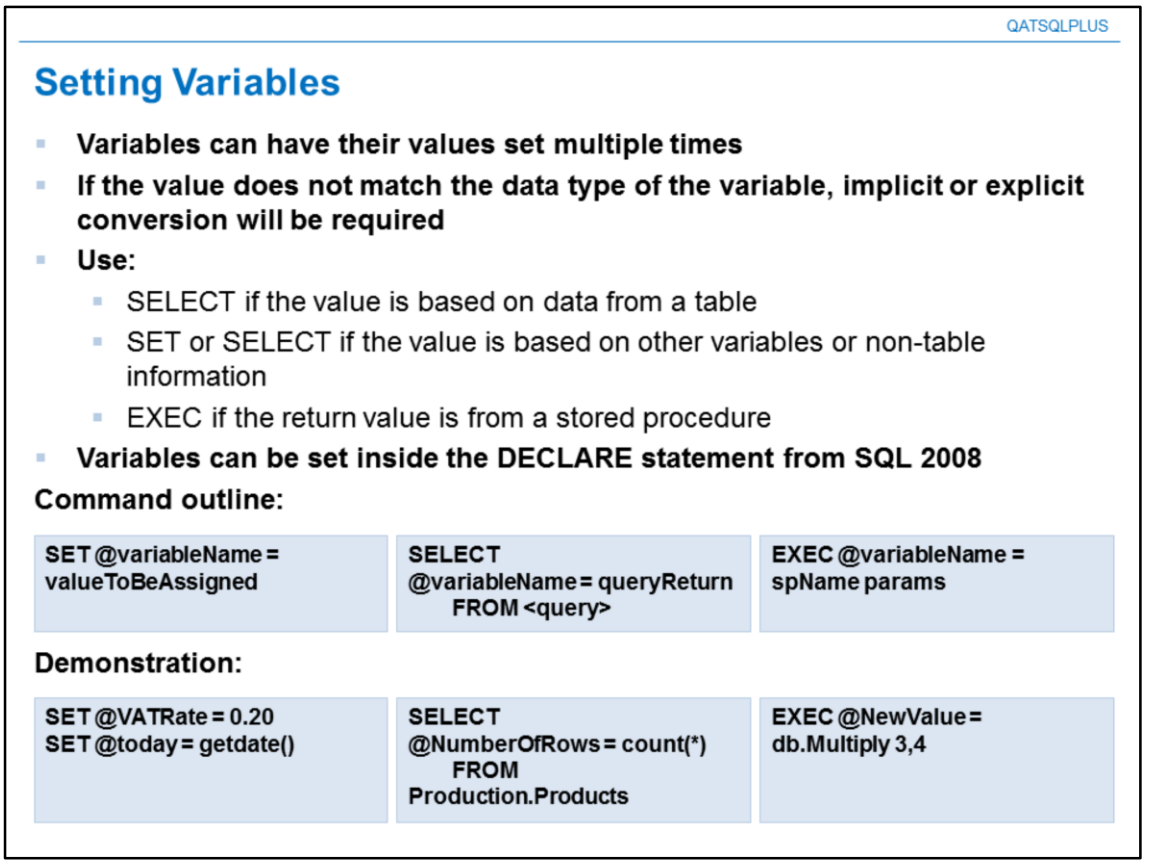
Declaring Variables (cont.)

Command outline:

```
DECLARE @variableName datatype
```

Demonstration:

```
DECLARE @Today DATETIME
DECLARE @Name VARCHAR(30)
DECLARE @Age INT
DECLARE @Salary MONEY
```



Setting Variables

- Variables can have their values set / reset multiple times, but the data type cannot be changed
- If the value does not match the data type of the variable implicit or explicit conversion will be required
- Variables can be set inside the DECLARE statement from SQL 2008
- Use:
    - SELECT if the value is based on data from a table

```
DECLARE @HighestPricedProduct MONEY = 0
SELECT @HighestPricedProduct = MAX(ListPrice)
   FROM production.Product
PRINT @HighestPricedProduct
```

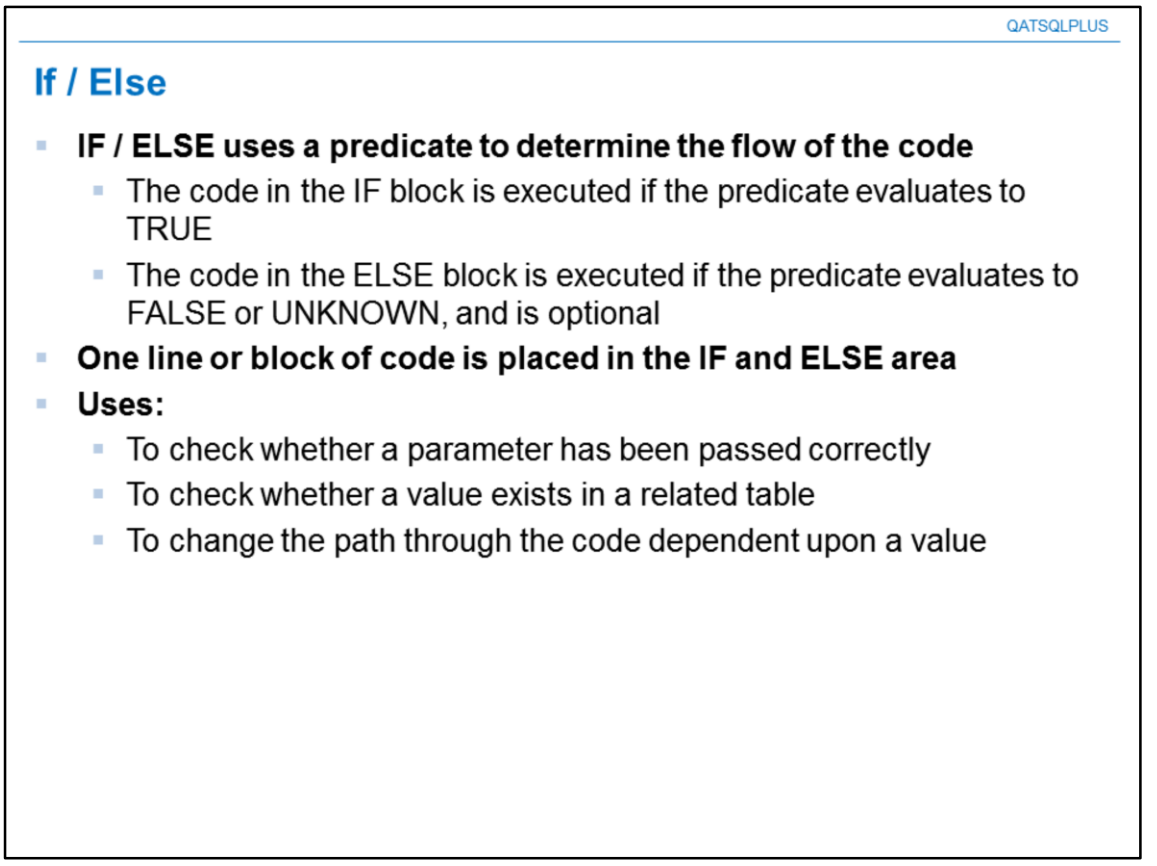
- Use:
  - SET or SELECT if the value is based on other variables or non-table information

```
DECLARE @Counter INT = 1
SET @Counter = @Counter + 1
SELECT @Counter = @Counter + 1
PRINT @Counter
```

  - EXEC if the return value is from a stored procedure

```
CREATE PROC dbo.CheckExists @ProductID INT AS
   BEGIN
      IF EXISTS (SELECT * FROM Production.Product WHERE
ProductID = @ProductID)
         RETURN 1
      ELSE
         RETURN 0
   END
GO

DECLARE @Exists INT
EXEC @Exists = dbo.CheckExists 3
PRINT @Exists
```



If / Else

- IF / ELSE uses a predicate to determine the flow of the code
  - The code in the IF block is executed if the predicate evaluates to TRUE
  - The code in the ELSE block is executed if the predicate evaluates to FALSE or UNKNOWN, and is optional
- One line or block of code is placed in the IF and ELSE area
  - If a second line of code is entered after the IF statement, then an ELSE cannot be used
  - To allow more than one line within the IF or ELSE sections then use BEGIN…END to produce a block of code
- Uses:
  - To check whether a parameter has been passed correctly

```
CREATE PROC dbo.CheckRating @R INT AS
  BEGIN
    IF (@R IS NULL)
      RETURN 0
    ELSE
      RETURN @R
  END
```

- Uses:
  - To check whether a value exists in a related table

```
CREATE PROC dbo.CheckExists @ProductID INT AS
   BEGIN
      IF EXISTS (SELECT * FROM Production.Product WHERE ProductID =
@ProductID)
         RETURN 1
      ELSE
         RETURN 0
   END
GO

DECLARE @Exists INT
EXEC @Exists = dbo.CheckExists 3
PRINT @Exists
```

  - To change the path through the code dependent upon a value

```
CREATE PROC dbo.ProductSearch @ProductID INT = NULL, @NameSearch
VARCHAR(30) = NULL AS
   BEGIN
      DECLARE @SearchString VARCHAR(30) = NULL

      IF (@ProductID IS NOT NULL)
  BEGIN
            SELECT * FROM Production.Product WHERE ProductID =
@ProductID
            RETURN
         END

      IF (@NameSearch IS NOT NULL)
  BEGIN
            SELECT * FROM Production.Product WHERE Name LIKE ('%' +
@NameSearch + '%')
            RETURN
         END

      SELECT * FROM Production.Product
   END
GO

EXEC dbo.ProductSearch @ProductID = 1          -- returns ProductID = 1
EXEC dbo.ProductSearch @NameSearch = 'Ball'    -- returns all rows where
Ball is in the name
EXEC dbo.ProductSearch       -- returns all rows
GO
```

If / Else

Command outline:

```
IF <condition>
    <some code>
ELSE
    <some other code>
```

Demonstration:

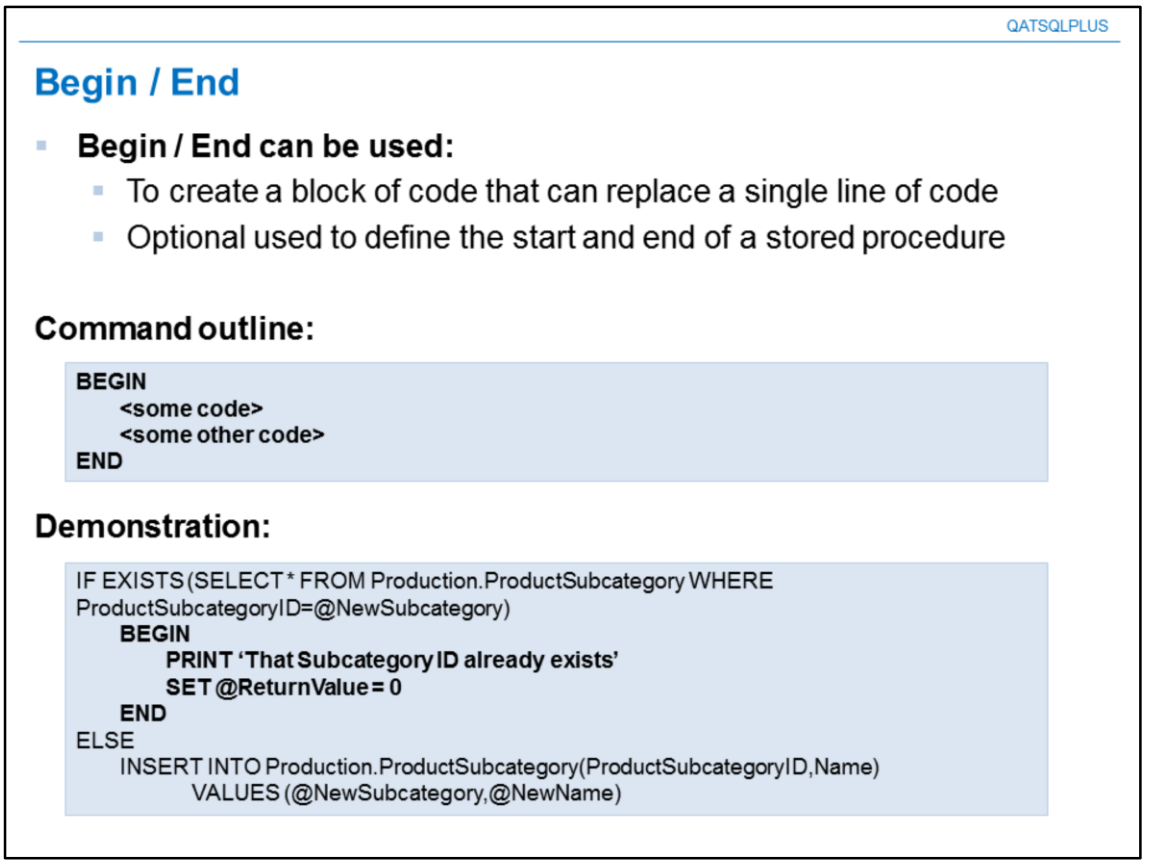
- Checking a value in a table

```
IF EXISTS (SELECT * FROM Production.ProductSubcategory WHERE
ProductSubcategoryID=@NewSubcategory)
    PRINT 'That Subcategory ID already exists'
ELSE
    INSERT INTO
Production.ProductSubcategory(ProductSubcategoryID,Name)
            VALUES (@NewSubcategory,@NewName)
```

If / Else (cont.)

- Single line of code only

```
IF EXISTS (SELECT * FROM Production.ProductSubcategory WHERE
ProductSubcategoryID=@NewSubcategory)
    PRINT 'That Subcategory ID already exists'
    SET @result = 0
ELSE – returns an error as more than one line of code between IF and ELSE
    INSERT INTO Production.ProductSubcategory(ProductSubcategoryID,Name)
        VALUES (@NewSubcategory,@NewName)
```



Begin / End

- Creates a block of code and can replace a single line of code, such as within an IF statement
- The code below will cause an error due to Else without If
  - The code would run successfully with the SET @ReturnValue = 0 line
  - This code may have been added later by error

IF EXISTS (SELECT * FROM Production.ProductSubcategory WHERE ProductSubcategoryID=@NewSubcategory)

PRINT 'That Subcategory ID already exists'

SET @ReturnValue = 0 ← **this line of code will run regardless of the result of the IF**

ELSE ← **error here due to more than one line between IF and ELSE**

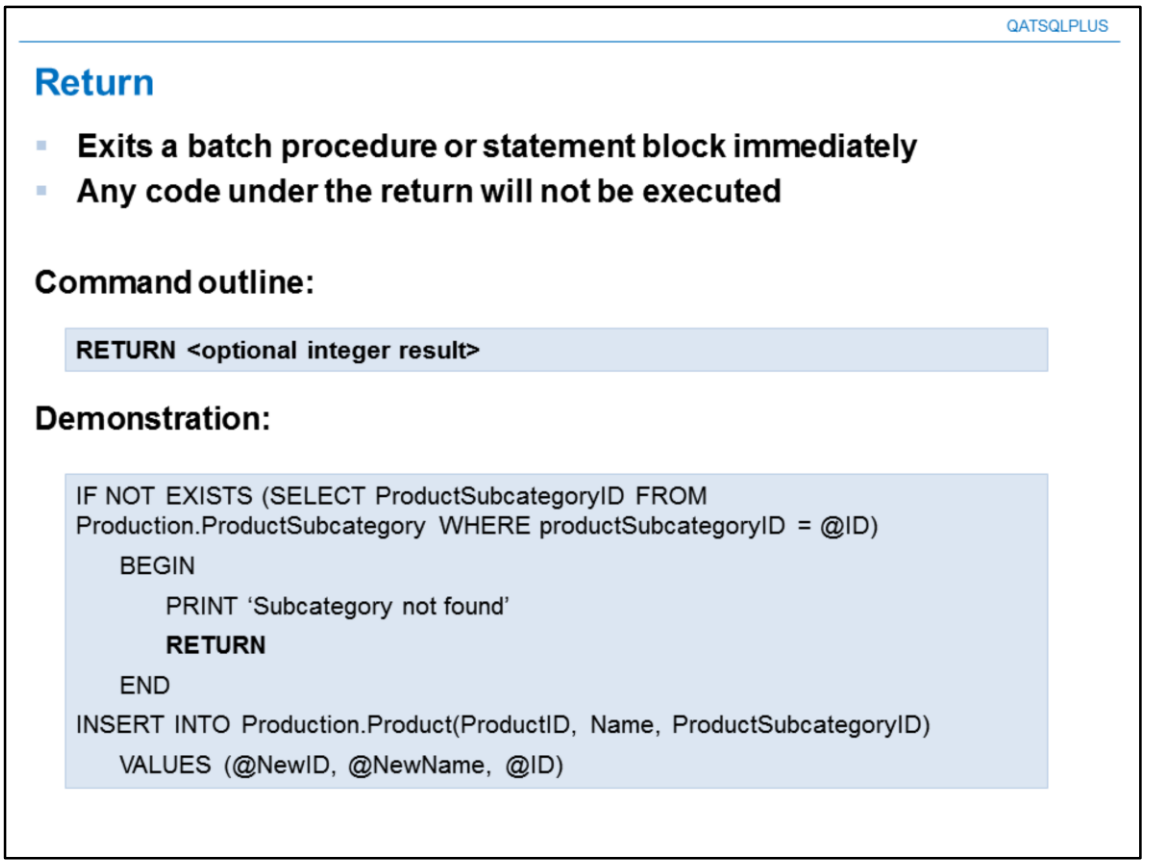
INSERT INTO Production.ProductSubcategory(ProductSubcategoryID,Name) VALUES (@NewSubcategory,@NewName

- Adding a BEGIN...END fixes this error

```
IF EXISTS (SELECT * FROM Production.ProductSubcategory WHERE
ProductSubcategoryID=@NewSubcategory)
    BEGIN
        PRINT 'That Subcategory ID already exists'
        SET @ReturnValue = 0
    END
ELSE
    INSERT INTO
Production.ProductSubcategory(ProductSubcategoryID,Name) VALUES
(@NewSubcategory,@NewName
```

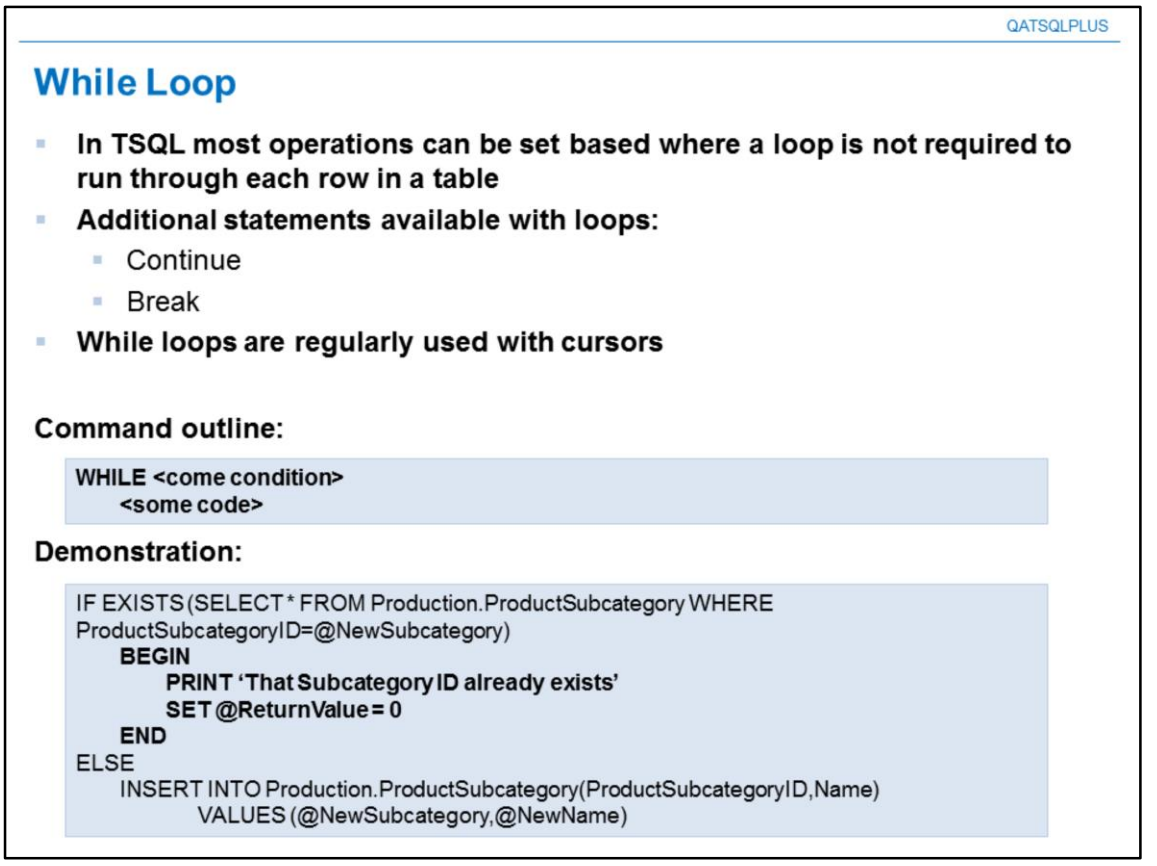
- Optional used to define the start and end of a stored procedure, although most developers follow this convention

```
CREATE PROC dbo.ProcName AS
    BEGIN
    …. Code
    END
GO
```



Return

- Exits a batch procedure or statement block immediately, which is useful when dealing with error checking
- Any code following the RETURN statement will not be executed
- Can be used within SQL batches to choose whether to continue with a batch or stop immediately



While Loop

- In TSQL most operations can be set based where a loop is not required to run through each row in a table
    - Using a loop to perform updates against a table is not good practice and will hamper performance of the server

Example

Given that the product has values 1 to 100 inclusive and the ListPrice needs to be increased by 10% for ListPrices under 1000 and 5% for ListPrices over or equal to 1000

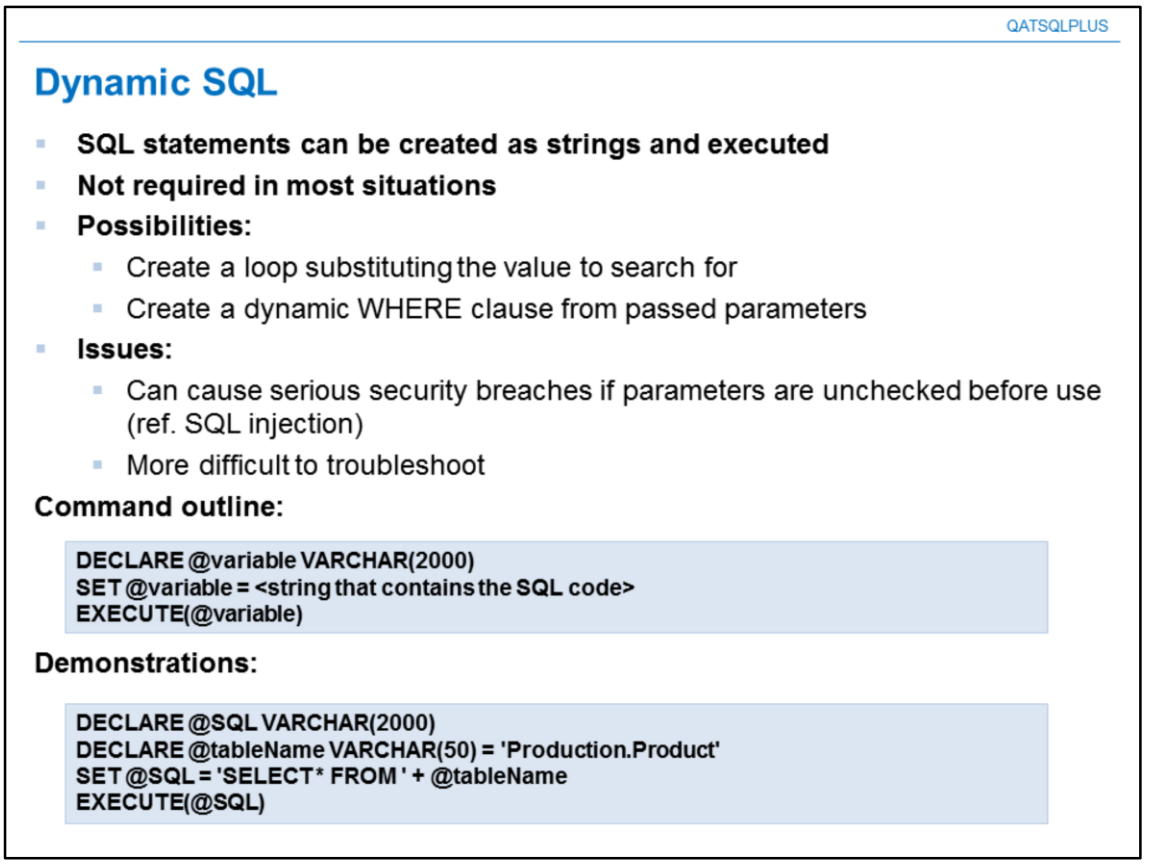
- Using loop (not best practice), slow processing of 100 rows with additional processing required for the loop checking, counter increment and 100 individual transactions

```
DECLARE @ID INT = 1
WHILE @ID <= 100
  BEGIN
    UPDATE Production.Product
      SET ListPrice = ListPrice * IIF(ListPrice < 1000, 1.10, 1.05)
      WHERE ProductID = @ID
    SET @ID = @ID + 1
  END
```

- Using SET processing, one transaction with 100 rows updated, optimised by SQL Server

```
UPDATE Production.Product
        SET ListPrice = ListPrice * IIF(ListPrice < 1000, 1.10, 1.05)
```

- Additional statements available within loops
  - Continue – Continues to the next WHILE iteration immediately
  - Break – Breaks out of a loop without checking the loop condition
- While loops are regularly used with cursors, where objects need to be processed and cannot be processed as a set
  - Cursors are covered later in the module



Dynamic SQL

- SQL statements can be created as strings and executed
- Not required in most situations
- Possibilities:
    - Create a loop substituting the value to search for, such as a cursor looping through multiple databases

Example

DECLARE @DatabaseName VARCHAR(255) = ‘Adventureworks2008R2’

DECLARE @SQL VARCHAR(2000)

SET @SQL = 'select '+""+ @DatabaseName + "" + ', name from ' + @DatabaseName + '.sys.objects where type="U" ‘

EXEC (@SQL)

- Create a dynamic WHERE clause from passed parameters although this may harm query performance as it does not allow for query plan reuse

- Issues:
  - Can cause serious security breaches if parameters are unchecked before use (ref SQL injection https://msdn.microsoft.com/en-us/library/bb669091(v=vs.110).aspx)
  - More difficult to troubleshoot as syntax and runtime errors cannot be reviewed before the execution of the dynamic SQL statements

Example

Given a parameter that should be a product name (not best practice)

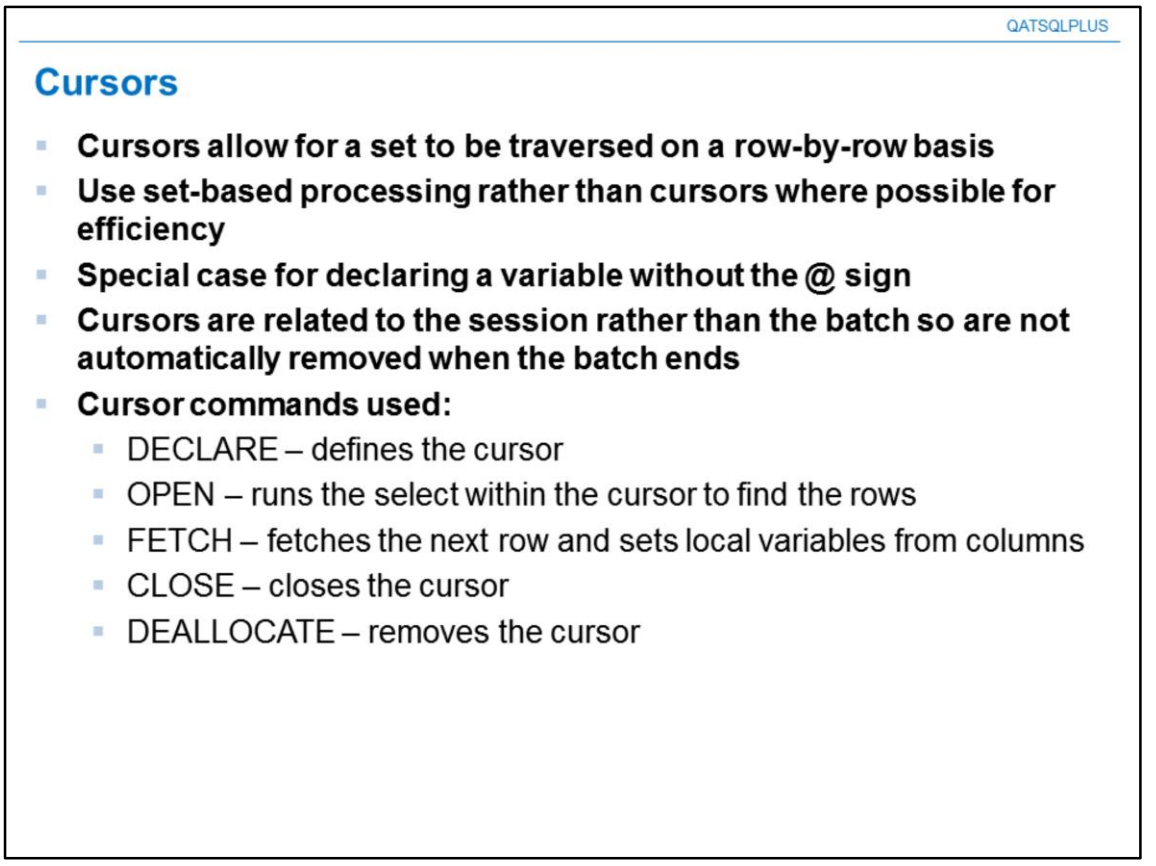
```
CREATE PROC dbo.getProductByName @ProductName VARCHAR(50) AS
   BEGIN
      DECLARE @SQL VARCHAR(300) = 'SELECT * FROM Production.Product'
      IF (@ProductName IS NOT NULL)
         SET @SQL = @SQL + ' WHERE Name = ' + ''' + @ProductName + '''
      PRINT @SQL
      EXEC (@SQL)
  END
GO

-- trials with parameters
EXEC dbo.getProductByName NULL                  -- all rows
EXEC dbo.getProductByName 'Bearing Ball'        -- Bearing ball row
EXEC dbo.getProductByName 'Any old rubbish'     -- no rows
EXEC dbo.getProductByName ' '' some thing '     -- error as extra quote in
                                                   string
```

Better practice:

```
CREATE PROC dbo.getProductByName @ProductName VARCHAR(50) AS
   BEGIN
      IF (@ProductName IS NOT NULL)
         SELECT * FROM Production.Product
      ELSE
  SELECT * FROM Production.Product WHERE ProductName = @ProductID
   END
GO

-- trials with parameters
EXEC dbo.getProductByName NULL                  -- all rows
EXEC dbo.getProductByName 'Bearing Ball'        -- Bearing ball row
EXEC dbo.getProductByName 'Any old rubbish'     -- no rows
EXEC dbo.getProductByName ' '' some thing '     -- no rows
```



Cursors

- Cursors allow for a set to be traversed on a row-by-row basis, although it is more efficient to use set-based processing rather than cursors where possible
- Unlike variables a cursor is declared, and referred to, without the @ sign
- Cursors are related to the session rather than the batch so are not automatically removed when the batch ends
    - An error will be returned if the same name is declared for a cursor the second time within a session, without the first cursor being deallocated
- Commands must be used:
    - **DECLARE** – defines the cursor within the session and specifies the source query and the columns returned
    - **OPEN** – runs the select within the cursor to find the rows within session
        - Depending upon the type of cursor defined locks may be placed on the rows / objects so that the cursor can complete without disappearing rows
    - **FETCH** – fetches the next row and sets local variables from columns, row-by-row before and during a while loop normally
        - Care must be take to fetch the next row within the loop or the cursor will stall on the current / first row
        - The fetch can return multiple columns at once
    - **CLOSE** – closes the cursor and releases any locks held
        - The cursor name is not released
    - **DEALLOCATE** – removes the cursor and released the cursor name

Cursors

Command outline:

```
DECLARE Cursorname CURSOR FOR <SELECT QUERY>
OPEN CursorName
FETCH NEXT FROM CursorName INTO <local variables>
WHILE @@Fetch_Status = 0
    BEGIN
    <some code using local variables>
    FETCH NEXT FROM CursorName INTO <local variables>
    END
CLOSE CursorName
DEALLOCATE CursorName
```

Cursors (cont.)

Demonstration:

```
DECLARE @DBName VARCHAR(100)
DECLARE @SQL VARCHAR(2000)
DECLARE @Results TABLE (DBName VARCHAR(100), TableName
VARCHAR(200))
DECLARE DBCursor CURSOR FOR SELECT Name FROM sys.databases
OPEN DBCursor
FETCH NEXT FROM DBCursor INTO @DBName
WHILE @@Fetch_Status = 0
    BEGIN
    SET @SQL = 'SELECT ' + '''' + @DBName + '''' + ', Name FROM [' +
@DBName +'].sys.tables'
    INSERT INTO @Results
        EXEC(@SQL)
    FETCH NEXT FROM DBCursor INTO @DBName
    END
CLOSE DBCursor
DEALLOCATE DBCursor
SELECT * FROM @Results
```

QATSQLPLUS

Exercise